Computer Programming C++

Howest, Fall 2014
Instructor: Prof. Jennifer B. Sartor
Jennifer.sartor@elis.ugent.be

About Me

- PhD at The University of Texas at Austin in August 2010
- Currently: professor at VUB & post-doctoral researcher at Ghent University
- I research how to make memory more efficiently managed, from the application on top of a Java virtual machine, to the operating system, then to hardware caches. I also do performance analysis of programs on modern multi-core machines.

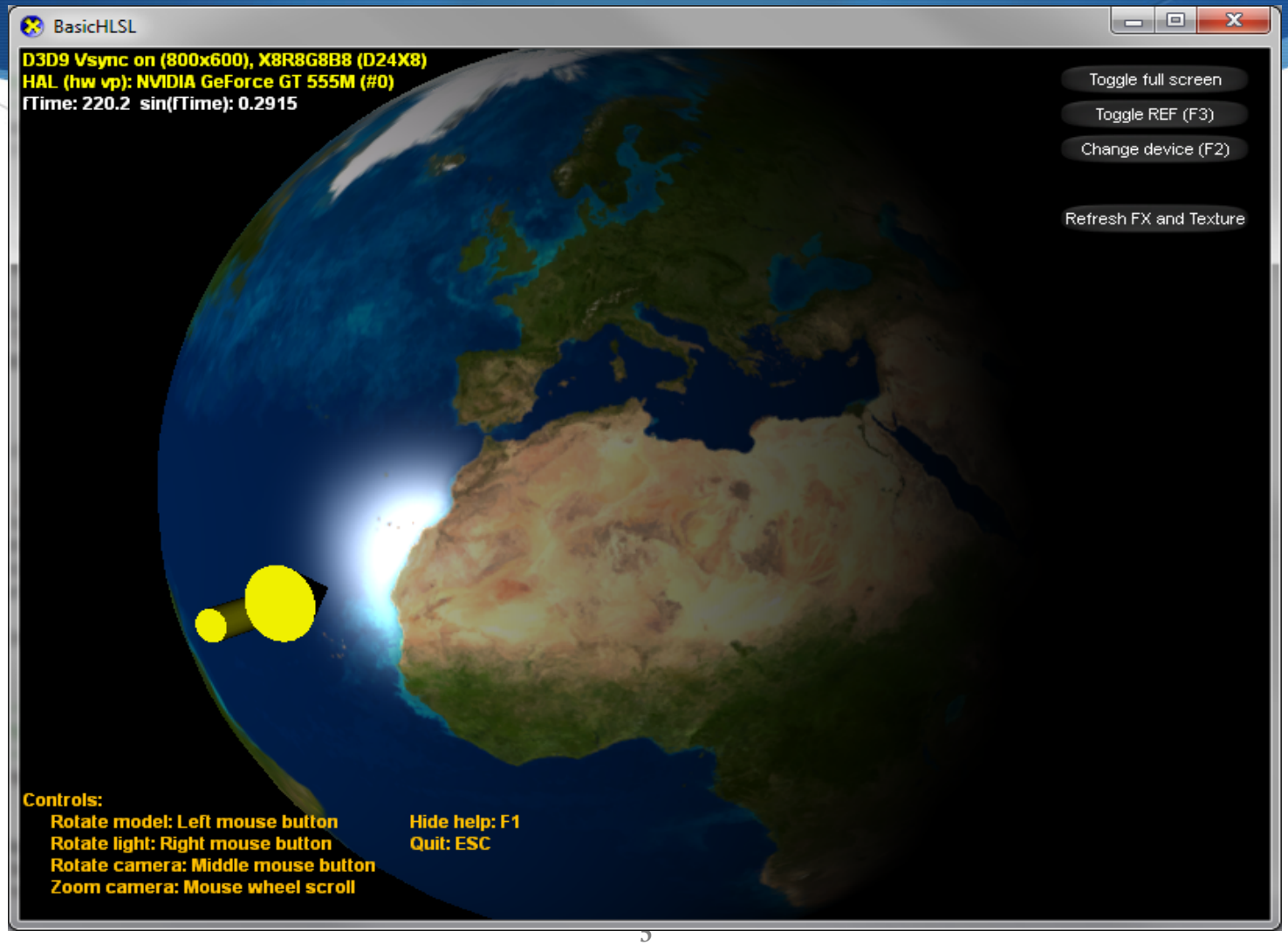
Whole Course

- Intro to C++ programming with me (6 classes, 3 hours each)
  - Jennifer.sartor@elis.ugent.be
- Intro to Graphics programming with (6 classes, 2 hours each)
  - C/C++-like language is used to program the GPU (like CUDA or OpenCL)
  - You will use C++AMP
  - Final project in graphics

GPU Final Project

- Textures used in video games are becoming larger and larger with sizes of up to 16k x 16k pixels. These textures are typically compressed to save disk space, e.g., using JPEG compression. Yet the GPU requires these textures to be in a compressed format called DXT. As a result, the game textures need to be transcoded from JPEG to DXT on the fly. The main goal of the project is to build a texture encoder which uses the massively parallel GPU to accelerate the DXT encoding steps.

Application for Final Project



Course Overview

- Intro to C++
- Good to have previous knowledge of object-oriented and procedural programming
- Website: http://users.elis.ugent.be/~jsartor/howest/c++Fall14.htm
- Communication will be through Minerva

Additional Info

- Books (optional)
  - Aan de slag met C++, Gertjan Laan
  - C++ Primer, Stanley B. Lippman,
  - C++: How to Program, Deitel & Deitel
- Grades will be based on programming assignments (80%) and one final test (20%)

Programming Assignments

- 4-5 programming assignments
  - Individual programming
  - In order to pass the class, you must submit all assignments, and they must compile and run (with provided test programs)
  - Programming style worth 15% of each assignment
  - 1 emergency late day (mulligan)

Microsoft Visual Studio

- General IDE (integrated development environment) to write, compile, debug, and run code
- You will use it for both C++ and C++AMP
- Download from Howest webpage
- Only runs on Windows platform

C++

- Extension of C, created by Bjarne Stroustrup in 1980s
- We will try to cover:
  - basic syntax, I/O, functions and argument passing,
  - arrays, references, pointers, classes, dynamic memory management,
  - classes and inheritance,
  - generic programming with templates
  - polymorphism with virtual functions and dynamic binding,

Similarities & Differences

- Look at basic.cpp for example C++ program
  - Operators, if/else, loops, commenting are the same as Java
- Variables are not by default zero-initialized!
- You need a main function:

```
int main() {
   …
   return 0; //success
}
```

Some I/O Basics

- At the top of a program
  - #include <iostream> //library you intend to use
  - Using declaration (which parts of library you will use), use either:
    1) using namespace std; //common (standard) parts of library
    2) using std::cin; using std::cout; //only these parts of library
- Input: int foo; cin >> foo;
- Output: cout << "bar " << foo;
- If you put "using std::endl" above, can use newline:
  cout << 5 << endl;

Functions

- Example:

```
int boxVolume(int side) {
    return side * side * side;
}
```

- Need to declare a function before it is used. If a function is defined later than it is used, provide **function prototype** at top:
  - int boxVolume(int);
  - int boxVolume(int side = 1); //can specify default parameters

C++ Compilation

- Compilation
  1) Preprocessor (expand things like #include <iostream>)
  2) Compiler – creates object (machine-language) code
  3) Linker – links object code with libraries, creates executable file
     myProgram.o + library.o = program.exe
- Preprocessor goes and finds library header file (iostream.h) with function prototypes for things you will use (cin, cout).
- Actual functions are defined in .cpp file (and .o object file) that linker will fetch.
- C++ compiler compiles for a specific machine type

Compiling on Command Line

- g++ basic.cpp (creates “a.out” executable)
- g++ -o program basic.cpp (“program” is executable)

  ./program
- g++ -c basic.cpp –o basic.o (create .o object file, compile but not link)
- Flags that are good practice
  - g++ -Wall -o program basic.cpp (print all warnings)
  - g++ **-Wall -Werror** -o program basic.cpp (treat warnings as compilation errors)

C++ Compilation

- Usually include files are called header files, are *.h and define function prototypes.
  - C++ libraries are usually in < >: #include <iostream> (compiler looks in standard library paths)
  - Header files **you** define are in " ": #include "myHeader.h" (compiler looks in current directory)
- Using declaration says exactly which function prototypes to include, or "namespace std" includes all common/standard ones.

How to Cast Safely?

- In Java:
  - double pi = 3.1415;
  - int num = (int) pi;
- C++ uses a **static cast**:
  - int num = **static_cast** <int> (pi);
  - Keyword, built into language

Storage Class: Static

- "static" is a keyword that helps program determine how long a variable will live
- Without "static" keyword, variable is alive inside it's code block
- With "static"
  - Exist from program begin to end
  - Can be global (outside a function or class) or local
  - Static local variables retain their value when function returns

Fun with Static Variables

- What does this print?

```
void func() {
    static int x = 0;
    x++;
    cout << x << endl;

}
```

```
int main() {
    func();
    func();
    func();
    return 0;

}
```

Storage of Variables

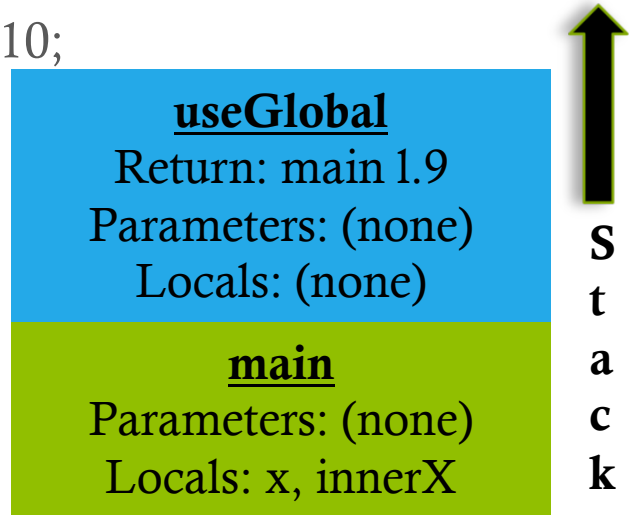
- Function call stack
  - Piece of memory allocated to manage information for calling and returning from a function
  - Each function called gets its own stack frame which holds information about the parameters passed, and the address to return to in the caller, and local variables
- Static and global variables are stored separately
- Later – dynamic memory and the heap, also separate

Storage and Scope

```
1.  int global = 1;
2.  void useStaticLocal();
3.  void useGlobal();
4.  int main() {
5.     int x = 5;
6.     { int innerX = 7;}
7.     useStaticLocal ();
8.     useGlobal();
9.  }
```

```
1.  void useStaticLocal () {
2.     static int num = 83;  //where?
3.     num++;
4.  }
5.  void useGlobal() {
6.     global *= 10;
7.  }
```

Statics and globals
global
num: useStaticLocal



Parameter passing

- 2 types
  - Pass-by-value
  - Pass-by-reference

Parameter passing

- 2 types
  - Pass-by-value
    - Argument copied
    - Caller and callee each have own copy
  - Pass-by-reference
    - Only 1 copy! Caller and callee share it
    - Beware – it can be modified by everyone

Pass-by-value

```
int squareByValue(int number) {
    return number *= number;
}
int main() {
    int x = 3;
    int x_squared = squareByValue(x);
    //what is x and what is x_squared?
}
```

Pass-by-reference

```
void squareByReference(int &number) {
    number *= number;
}
int main() {
    int x = 3;
    squareByReference(x);
    //what is x?
}
```

Pass-by-reference

- void myFunction(int &x);
  - x is a reference to an int.
- Be careful – you are giving callee method power to change your variable
- To save copying space, but protect your variable, use **const**
  - void myFunction(const int &x);
  - Now myFunction cannot modify x.

References as Aliases

```
int count = 1;

int &cRef = count;

cRef++;

//count is ?
```

- Reference variables must be initialized in their declaration and cannot be reassigned later.

Returning References

- Returning references from a function is dangerous
- Variable declared on stack cannot be returned
- Can return static variable, but might not be what you want
- Dangling reference = reference to undefined variable

Arrays!

- Indexed data structure
  - Starts at zero!
- How do we declare an array?

Arrays!

- Indexed data structure
  - Starts at zero!
- How do we declare an array?
  - *type arrayName[arraySize];*
  - Ex: int array[5];

Array Initialization

- Loop
- Initializer list
- Use **const** array size

Array Initialization

- Loop

```
int array[5];
for (int i = 0; i < 5; i++) {
    array[i] = i;
}
```

- Initializer list
- Use **const** array size

Array Initialization

- Loop
- Initializer list
  - int array[5] = {99, 88, 77, 66, 55};
  - int array2[5] = {}; //what does this do?
  - int array[] = {44, 33, 22, 11};
- Use **const** array size

Array Initialization

- Loop
- Initializer list
- Use **const** array size

  const int arraySize = 10;

  int array[arraySize]; //what does this array contain?
  - const variables must be initialized when declared, are constant
  - Only constants can be used to declare size of local (stack) and static arrays

Differences from Java

- No automatic ".length" for arrays
- No guaranteed compiler array bounds checks – if you go outside [0 through (arraySize-1)], undefined behavior
- Arrays are always contiguous in memory
- Arrays are not by default zero-initialized

Character Arrays

- char string1[] = “hello”;
- What is the size of the array above?

Character Arrays

- char string1[] = “hello”;
- What is the size of the array above? **6**
- Char arrays are terminated with null character!
- char string1[] = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};

```
for (int i = 0; string1[i] != ‘\0’; i++) {
    cout << string1[i] << ‘ ‘;

}
```

Strings

- C++ does have string type
  - #include <string>
  - string hello = “hello”;
- Some useful string functions:
  - hello.data( ); or hello.c_str(); //get string’s character array
  - hello.length(); //get length
  - char oneChar = hello[1]; //can index strings
  - string wstr = “world”;
    hello.append(wstr, 0, wstr.length()); //append to get “helloworld”
  - char chArray[10];
    wstr.copy(chArray, wstr.length(), 0); //copy wstr string into array

Input with Char Arrays

- char string2[20];
- cin >> string2;
  - cin reads in a string (until whitespace) and appends null character to end
  - Make sure input from user <= 19 characters, otherwise error
- For a line at a time:
  - cin.getline(string2, 20);
  - string myStr; getline(cin, myStr);

Passing Arrays to Functions

```
void modifyArray(int [], int);

int main() {
    const int arraySize = 5;
    int a[arraySize] = {0, 1, 2, 3, 4};
    modifyArray(a, arraySize);
    return 0;

}
```

```
void modifyArray(int b[],
    int arrSize) {
    for (int i = 0; i < arrSize; i++) {
        b[i] *= 2;
    }

}
```

Passing Arrays to Functions

- Arrays are passed by reference
- Name of array is the address in memory of the 1st element
- Need to pass size too – unlike Java
- Use const to make sure function can't change array
  - void cannotModifyArray(const int b[]);

Static Local Arrays

```
void staticArrayInit();

int main() {
    staticArrayInit();
    staticArrayInit();
    return 0;
}
```

```
void staticArrayInit( void ) {
    static int array1[3];
    for (int i = 0; i < 3; i++) {
        arrays1[i] += 5;
    }

} //what if array is not static?
```

Multidimensional Arrays

- int array[2][3] = {{1, 2, 3}, {4, 5, 6}};
- int array[2][3] = {1, 2, 3, 4};
- int array[2][3] = {{1, 2}, {4}};
- Different from Java – contiguous in memory
- 2nd dimension needs to be known when passing to a function

Multidimensional Arrays

- int array[2][3] = {{1, 2}, {4}};
- Different from Java – contiguous in memory
  - Conceptually:

[0][0] = 1	[0][1] = 2	[0][2] = 0
[1][0] = 4	[1][1] = 0	[1][2] = 0

  - Actual layout:

[0][0] = 1	[0][1] = 2	[0][2] = 0	[1][0] = 4	[1][1] = 0	[1][2] = 0

Passing Multi-D Arrays

```
void printArray( const int[][3],
                 int numRows );

int main() {
    int array1[2][3] = {1, 2, 3, 4};
    printArray(array1, 2);
    return 0;

}
```

```
void printArray( const int[][3],  //why?
                 int numRows) {
    for (int i = 0; i < numRows; i++) {
        for (int j = 0; j < 3; j++) {
            cout << a[i][j] << ' ';
        } cout << endl;
    }
}
```

2D as 1D array

- int array1[2][3] = {};

[0][0] = 0	[0][1] = 0	[0][2] = 0	[1][0] = 0	[1][1] = 0	[1][2] = 0

- // to access array1[1][0] – we need to skip over 1st row then go over to element 0 in second row
- //number of entries per row = number of columns
- array1[3 * 1 + 0] == array1[1][0];
- //formula: numColumns * 1stIndex + 2ndIndex

typedef

- **typedef** is a keyword that declares synonyms (aliases) for previously defined data types
- Does not create a data type, it creates a type name (usually shorter, simpler) that maybe be used in the program
- **typedef** unsigned long int ulint;
- ulint myNum;
- size_t is a typedef for unsigned int (used for string's length())

sizeof Operator

- **sizeof** does exactly what you'd expect – give it a variable or type, it will return the size of it in bytes.
- return type: not int, but size_t (unsigned int)

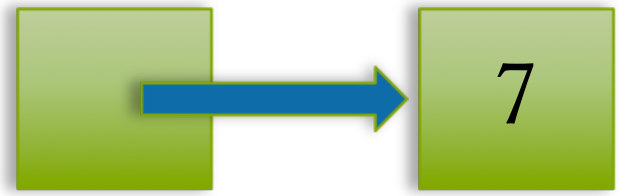
```
int x = 5;

cout << sizeof x << endl;  //can omit parens with variable

cout << sizeof( int ) << endl;
```

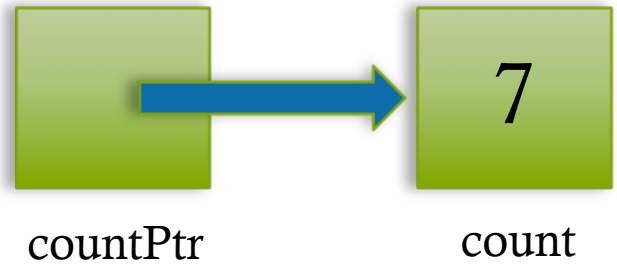
Pointers

- Mysterious, but very powerful.
- int *countPtr, count; //what are types of each variable?
- count = 7;
- countPtr = &count;
- & means “obtain memory address”
- countPtr indirectly references count



Pointer Operators

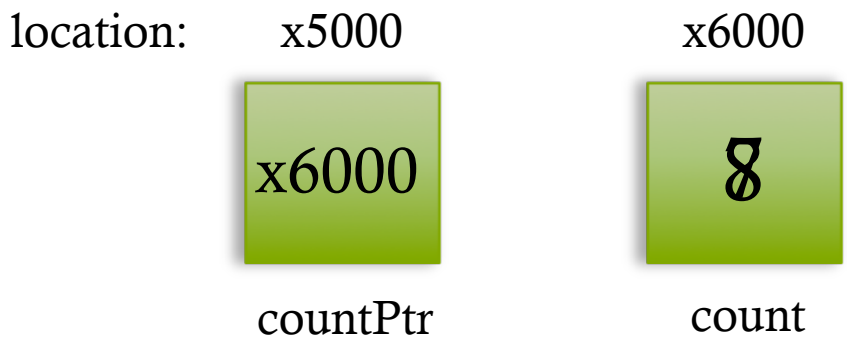
- Similar to references
- count = 7;
- countPtr = &count;
- *countPtr == count == 7;



- * is indirection or dereferencing operator. * returns synonym for object to which operand points.
- & and * are inverses

Pointers

- int *countPtr, count;
- count = 7;
- countPtr = &count;
- *countPtr++;
- countPtr indirectly references count, *countPtr is called “deferencing a pointer”



Pointer Operators

- * is indirection or dereferencing operator.
- Pointer is undefined when created – can be set to 0 or NULL
- Dereferencing an uninitialized or NULL pointer is BAD!
  - What if we did (*count)?
  - Or int *countPtr; then (*countPtr) ?

count

Pointers vs. References

- **Differences**
  - In reference declaration (int &cRef = count;), "&" is part of type, it is not an operation (as with pointers)
  - References have to be initialized at declaration time
- void func_ptr(int *pi) {*pi = 5;}
- void func_ref(int &ri) {ri = 6;}
- int num; int *p = # int &r = num;
- func_ptr(&num);
- func_ref(num); //We are passing parameters by… what?

Pointer Example

```
void cubeByReferenceWithPointer(int *nPtr) {
    *nPtr = *nPtr  *  *nPtr  *  *nPtr;

}

int main() {
    int number = 5;
    cubeByReferenceWithPointer(&number);
    cout << number;
    return 0;

}
```

Arrays are Just Pointers

int arrayName[5] = {};

- "arrayName" is constant pointer to start of array
- arrayName == &arrayName[0];
- void modifyArray(int [], int) == void modifyArray(int*, int);
- Array parameter translated by the compiler to be int *. So 2nd function above has to know whether it receives array or int pointer.

sizeof Array vs. Pointer

```
size_t getSize(double *);

int main() {
    double array[20];
    cout << sizeof(array) << endl;
    cout << getSize(array) << endl;
    cout << (sizeof(array) / sizeof(double)) << endl;  //array length
    return 0;

}
```

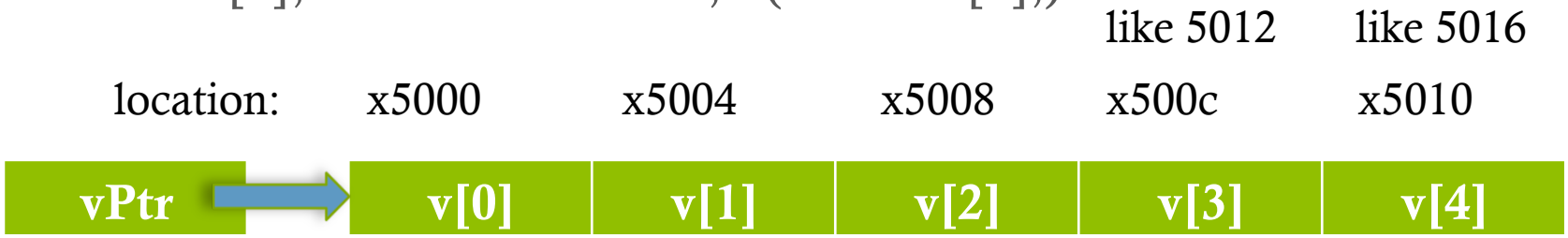
```
size_t getSize (double *ptr) {

    return sizeof(ptr);

}
```

Parameter Arithmetic & Arrays

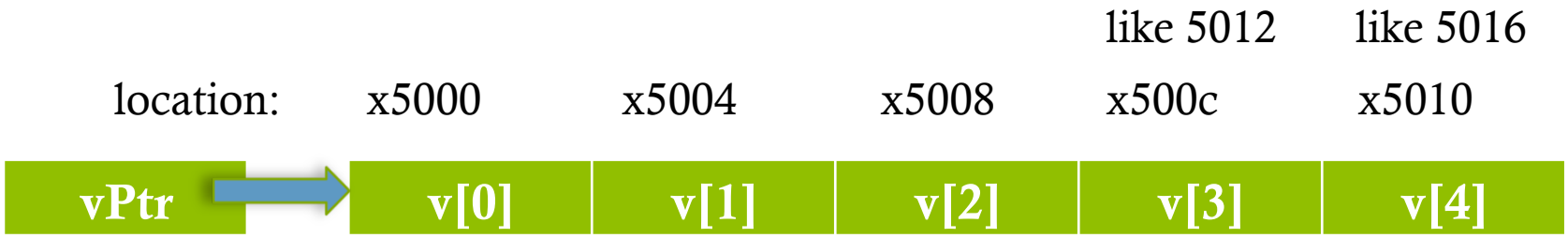
- int v[5]; int *vPtr = v; (or = &v[0];)



- vPtr += 2; //goes from x5000 to ?

Parameter Arithmetic & Arrays

- int v[5]; int *vPtr = v; (or = &v[0];)



- vPtr += 2; //goes from x5000 to x5008
- Pointer arithmetic depends on type of pointer
- cout << (vPtr – v) << endl; //what is this?

Parameter Arithmetic & Arrays

- int v[5]; int *vPtr = v; (or = &v[0];)
- v[3] == * (vPtr + 3) == * (v + 3) == vPtr[3]
- vPtr + 3 is the same as &v[3]
- Array names cannot be modified in arithmetic expressions because they are constant.

Void*

- void* voidPtr;
- void* is generic pointer, it can point to any type, but can't be dereferenced.
  - Cannot do (*voidPtr) (even if initialized) – why?
- All pointer types can be assigned to a pointer of type void* without casting. void* pointer **cannot** be assigned to pointer of other type without casting.
  - void* voidPtr = whateverPtr; //assigning specific to general
  - int* intPtr = (int*) voidPtr;//assigning general to specific – need cast

Arrays and Pointers

```
void myPrint(const char *);

int main() {
    char *phrasey = “C++Fun”;
    myPrint(phrasey);
    return 0;

}
```

```
void myPrint(const char * s1) {

    while ((*s1) != ‘\0’) {
        cout << *s1;
        s1++;
    }

}
```

Arrays and Pointers

```
void copy1(char*, const char *);

int main() {
    char phrase1[10];
    char *phrase2 = “Hello”;
    copy1(phrase1, phrase2);
    cout << phrase1 <<  endl;
    return 0;

}
```

```
void copy1(char * s1,

           const char * s2) {
    for(int i =0; s2[i] != ‘\0’; i++) {
        s1[i] = s2[i];
    }

}
```

Arrays and Pointers

```
void copy2(char*, const char *);

int main() {
    char phrase3[10];
    char *phrase4 = "GBye";
    copy2(phrase3, phrase4);
    cout << phrase3 <<  endl;
    return 0;

}
```

```
void copy2(char * s1,
           const char * s2) {
    for(; *s2 != '\0';
          s1++, s2++) {
          *s1 = *s2;
    }

}
```